SAND REPORT

SAND2002-4016
Unlimited Release
Printed -December 2002

SIERRA Framework Version 3: h-Adaptivity Design and Use

James R. Stewart and H. Carter Edwards

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Sandia National Laboratories

SAND2002-4016
Unlimited Release
Printed December 2002

SIERRA Framework Version 3: *h*-Adaptivity Design and Use

James R. Stewart and H. Carter Edwards
Production Computing/SIERRA Architecture Department
Engineering Sciences Center
Sandia National Laboratories
Box 5800
Albuquerque, NM 87195-0827

Abstract

This paper presents a high-level overview of the algorithms and supporting functionality provided by SIERRA Framework Version 3 for *h*-adaptive finite-element mechanics application development. Also presented is a fairly comprehensive description of what is required by the application codes to use the SIERRA *h*-adaptivity services. In general, the SIERRA framework provides the functionality for hierarchically subdividing elements in a distributed parallel environment, as well as dynamic load balancing. The mechanics application code is required to supply an *a posteriori* error indicator, prolongation and restriction operators for the field variables, hanging-node constraint handlers, and execution control code. This paper does not describe the Application Programming Interface (API), although references to SIERRA framework classes are given where appropriate.

Acknowledgement

We would like to thank Lee Taylor for providing the figures and much of the text in Section 4.2, and Kevin Copps for providing Fig. 4.1. We also thank James Overfelt and Nathan Crane for reviewing the manuscript. Finally, we thank Rhonda Reinert of Technically Write, Inc. for providing substantial technical editing and formatting assistance.

Contents

Figures

Intentionally Left Blank

1 Overview

Support for *h*-adaptivity is one of the advanced services provided by SIERRA. This paper describes at a high level the algorithms and supporting functionality available in SIERRA Framework Version 3 for developing *h*-adaptive finite-element mechanics applications. We also describe what is needed by the mechanics code for using the *h*-adaptivity services. This paper is not intended to be a reference for the Application Programming Interface (API). We recommend consulting the SIERRA header files for those details. The primary interface to this set of services is found in the `Fmwk_HAdapt` class (see `Fmwk_HAdapt.h`). References to other SIERRA classes are also given in this paper where appropriate. A description of the core SIERRA Framework Version 3 theory and design is given in Ref. 1. It is assumed herein that the reader has some familiarity with the contents of that document.

Adaptivity is defined to be the process of locally increasing the computational mesh resolution in order to achieve a desired solution accuracy more efficiently. Adaptivity can also be used to achieve a *more* accurate solution for a fixed amount of resources (e.g., computer memory). Without adaptivity, the only alternative (for a given solution method) is to *globally* increase the mesh resolution. Depending on the particular problem, this can result in a large number of unneeded degrees of freedom, in turn leading to a high solution cost. The local increase of mesh resolution in an adaptive computation can be achieved in various ways. In *h*-adaptivity, the mesh is refined by locally generating smaller elements, i.e., by decreasing the size *h* of the elements that contribute the largest errors. Other types of adaptivity include *p*-adaptivity (locally increasing the order of the finite element shape functions), *hp*-adaptivity (a combination of *h* and *p*-adaptivity), and *r*-adaptivity (relocating nodes to locally achieve smaller elements).

There are many different steps needed for an adaptive computation, and there is a clear separation as to which of these steps are "framework" and which are "application." A general overview of the steps composing the adaptive algorithm (applicable to most applications) is given in Fig. 1.1.

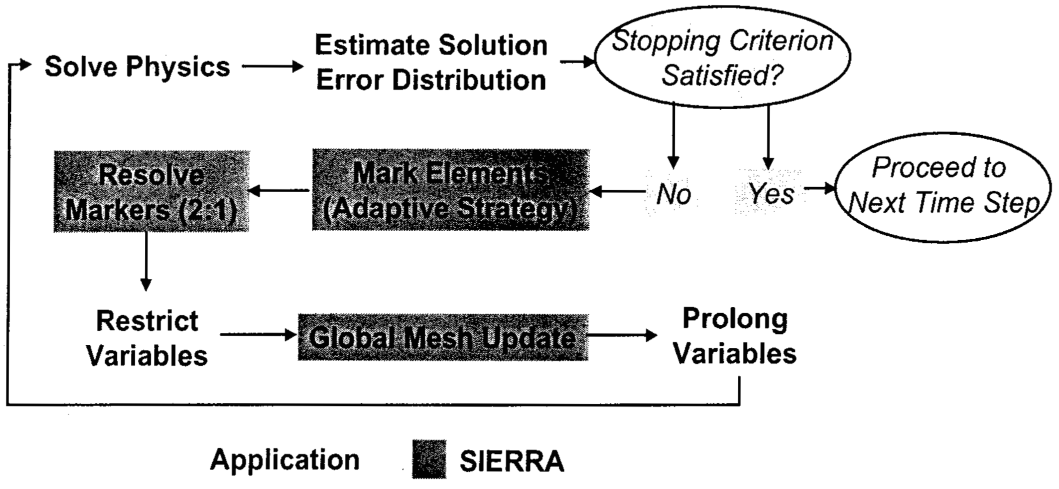
Solve Physics
Estimate Solution Error Distribution
Stopping Criterion Satisfied?
No
Yes
Proceed to Next Time Step
Mark Elements (Adaptive Strategy)
Resolve Markers (2:1)
Restrict Variables
Global Mesh Update
Prolong Variables
Application
SIERRA

Figure 1.1. General adaptive algorithm applicable to most applications. Responsibilities of the framework and application are shown. The marking of elements (the "adaptive strategy") can be implemented either by the framework (for a mechanics-independent strategy) *or* by the application (for a mechanics-dependent strategy).

In general, SIERRA provides the parallel mesh refinement and unrefinement steps, as well as dynamic load rebalancing. Those services are all part of the "Global Mesh Update" step depicted in Fig. 1.2.

SIERRA provides standard h-refinement of meshes, i.e., elements are subdivided into a given number of topologically compatible children. Upon unrefinement, the child elements are removed and the original parent element is restored. The application "marks" each element for refinement, unrefinement, or neither, based on some error criterion. The framework then takes over and carries out the requested mesh adaptation.

Utilizing other SIERRA services, application-specific functionality such as the error indicator and prolongation (and restriction) operators for the field variables can easily be plugged in. It may also be desired to call the error estimator *without* doing any adaptivity. This flexibility is completely controlled by the application code.

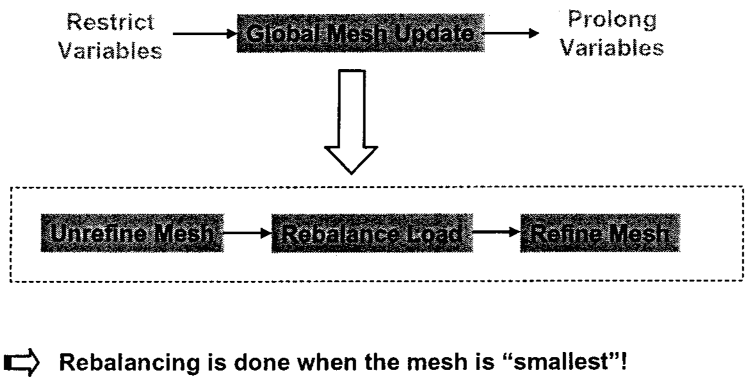

Figure 1.2. Composition of the "Global Mesh Update" service. This service is composed of three steps: mesh unrefinement, followed by dynamic load rebalancing, followed by mesh refinement.

The application developer controls how many times the global mesh update is executed. For transient calculations, this sequence can be executed multiple times within a time step. A refined element is referred to by the *level* of refinement, which is the number of (net) refinement steps required to generate that element. To control mesh gradation, the framework enforces a maximum 2:1 ratio of refinement levels of adjacent elements (this step is shown in Fig. 1.1). This might cause elements that have not been designated for refinement by the application to be refined anyway.

The 2:1 ratio is globally enforced, i.e., refinement on a given processor may trigger refinement on other processors. The constrained *hanging* nodes that appear along a refinement boundary must be appropriately handled by the application code. Importantly, refinement always takes precedence over unrefinement. For example, an element that has been marked for unrefinement would not be unrefined if doing so would fail to maintain the maximum 2:1 refinement-level ratio.

In the following sections we describe in more detail both the framework components of adaptivity and the responsibilities of application codes.

2 Framework Adaptivity Services

As indicated in Fig. 1.1, SIERRA provides the global mesh update as well as the 2:1 refinement-level enforcement. The global mesh update, as shown in Fig. 1.2, consists of mesh unrefinement, dynamic load rebalancing, and mesh refinement (in that order). Each of these steps is described in more detail in this section. In addition, SIERRA provides a default mechanics-independent *h*-adaptive strategy. Through the use of C++ virtual methods, SIERRA allows an application code to override this default. For example, an error-based *h*-adaptive strategy may be developed to work in conjunction with a particular error estimator. SIERRA does *not* provide a default *a posteriori* error indicator or error estimator. These are mechanics specific and must be supplied by the application code. The default *h*-adaptive strategy is also described in this section.

2.1 Parallel Mesh Refinement and Unrefinement

Mesh refinement is carried out by hierarchically subdividing elements. The newly created, refined elements (the *children*) become active in the subsequent computation. However, the coarse *parent* element is retained in the data structure and becomes inactive. An element may only be unrefined back to level 0. That is, if the element does not have children, it cannot be unrefined, implying that a mesh can get no coarser than the *genesis* mesh. The genesis mesh is the original mesh read from the input file. Once an element has been unrefined, each of the child elements (along with any faces, edges or nodes that were used only by the children) is deleted from the mesh data structure, and the previously inactive parent element becomes active again.

Remark

An alternative strategy would be to allow arbitrary unrefinement to any mesh coarseness. For example, unrefinement (and refinement) could be implemented using local mesh manipulations (see, for example, Ref. 2). While this is a desirable feature, it is difficult to implement for certain element topologies such as hexahedra. One could also completely remesh the domain, but this is prohibitively expensive in three dimensions, and perhaps impossible to make robust in parallel. SIERRA is driven by the requirement to support many different types of element topologies including hexahedra, tetrahedra, pyramids, beams, and wedges (possibly occurring simultaneously in the mesh). This requirement led to the design decision to retain parent elements in the data structure, which allows refinement and unrefinement to topologically compatible child and parent elements, respectively, to be easily obtained in a parallel environment.

The refinement process is depicted in Fig. 2.1 for an initial mesh of four quadrilateral elements. When an application marks an element for refinement (denoted by "R" in the figure), the framework takes over and produces the desired refinement. Recall that the adaptation can take place iteratively. If a child element is subsequently refined as shown in the middle picture in Fig. 2.1, the framework also refines adjacent (unmarked) elements to enforce the 2:1 maximum refinement-level ratio. As shown in the picture on the right, this enforcement may trigger a communication step to force elements that may be on an adjacent processor to also be refined. The resulting adaptive mesh is independent of parallel decomposition!

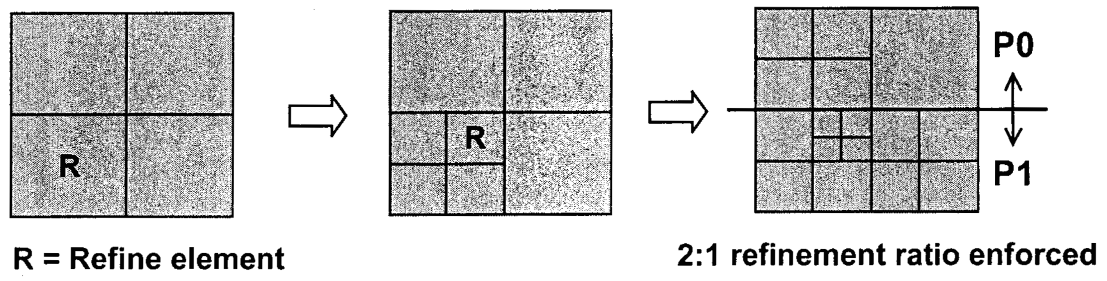

Figure 2.1. Adaptive mesh refinement process including global enforcement of the 2:1 maximum refinement-level ratio.

Although not available as of the release of SIERRA Framework Version 3.01, the capability for snapping new nodes to the actual geometry is planned. This capability will be used for new nodes placed on nonplanar surfaces or material interfaces, and is critical for accurately representing the geometry as the mesh is refined.

The unrefinement process is shown in Fig. 2.2. When an application marks each child of a particular parent element for unrefinement (denoted by "U" in the figure), the framework takes over and produces the requested unrefined mesh.

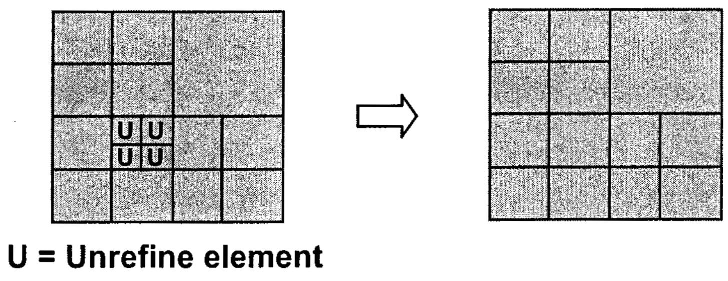

Figure 2.2. Adaptive mesh unrefinement process.

Unrefinement can occur only if the following criteria hold:

- All of a parent's child elements have been marked for unrefinement.
- Unrefinement will not cause the maximum-allowable 2:1 refinement-level ratio to be exceeded.

These cases are shown in Fig. 2.3.

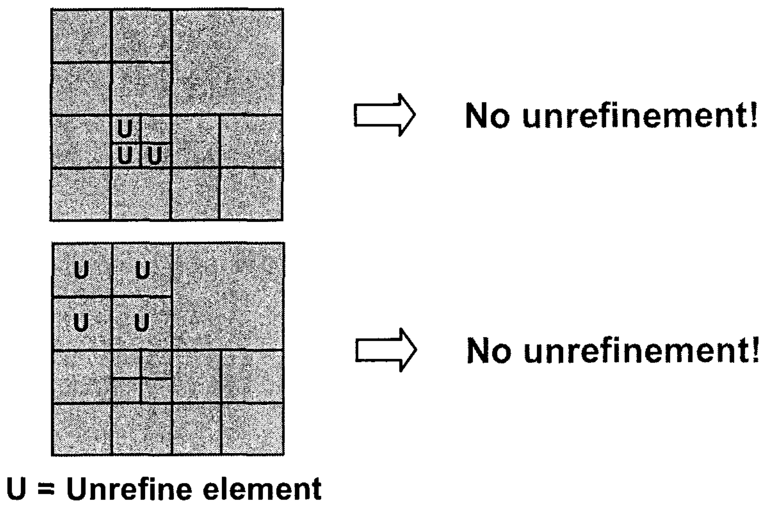

Figure 2.3. Cases that violate the unrefinement requirements. In the first case, not all of the parent's children are marked. In the second case, unrefinement would break the 2:1 maximum refinement-level ratio. In either case the framework ignores the unrefinement requests.

SIERRA uses a refinement template to define the topologies of child elements. In general, these topologies must be compatible with the parent topology, in the sense that the resulting shapes on the boundaries of parent elements must allow for (child) mesh continuity and validity across those boundaries. In the simplest cases, a hexahedron is subdivided into eight child hexahedrons, a tetrahedron is subdivided into eight child tetrahedrons, and a wedge (three quadrilateral faces and two triangular faces) is subdivided into eight child wedges. The two-dimensional examples of a quadrilateral and a triangle are shown in Fig. 2.4. In all of these examples, the child mesh is formed by connecting the midpoints of the parent edges to form a valid subdivision of the parent element.

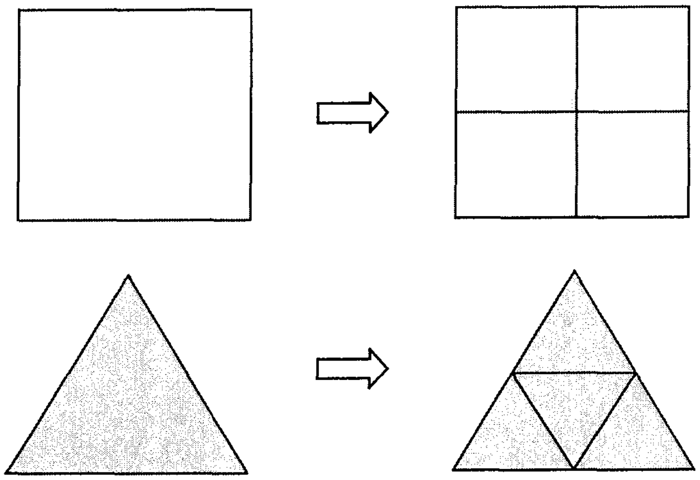

Figure 2.4. Refinement templates as defined for the quadrilateral and triangle elements.

2.2 Dynamic Load Rebalancing

Following the mesh unrefinement step and preceding the mesh refinement step, the application can request that the framework dynamically rebalance the mesh among the processors. As indicated in Fig. 1.2, the steps are done in this order so that rebalancing occurs when the mesh is smallest (to minimize the communication overhead). Since rebalancing is an option, the application can turn it on or off at any time. For example, the application could choose to only rebalance every other refinement iteration. Alternatively, the application could *compute* a measure of "balance," then turn on rebalancing only if it is needed.

The application is responsible for defining the computational load on each element (the *element load measure*) which is then used by the partitioning algorithm. The framework provides methods for registering the element load measure variable on each element (see `Fmwk_Region.h`). The default value for the load measure is 1.0. Since load rebalancing occurs *before* mesh refinement, the load measure on to-be-refined elements must be adjusted to account for the load following refinement. This adjustment is handled by the framework. Before calling the partitioning algorithm, the framework multiplies each to-be-refined element (i.e., each currently active future parent) by the number of children that parent will produce. It is assumed that the element load measure of the children will be the same as that for the parent.

The determination of a new, load-balanced partition of the mesh is carried out by the Zoltan dynamic load-balancing library [3]. The framework then uses this partition to redistribute the mesh among the processors. The mesh-partitioning algorithm is limited to those available in Zoltan, and can be specified by the application. The SIERRA interface to Zoltan is given in the `Fmwk_Rebalance` class (see `Fmwk_Rebalance.h`).

Remark

The complete functionality for dynamic load rebalancing without using *h*-adaptivity is available to the application through the Fmwk_Rebalance interface. For example, this might be useful for calculations with chemical reactions where the workload in each element varies depending on the number of chemical species present.

The framework provides an option for restricting the scope of the rebalancing to *genesis extent* only. The interface to this option is given in the `Fmwk_HAdapt` class (see `Fmwk_HAdapt.h`). The description and purpose of this option are described next.

Rebalancing with Genesis-Extent-Only Option

The genesis extent refers to a mesh extent (i.e., a set of mesh objects) as defined in SIERRA (see, e.g., `Fmwk_MeshExtent.h`). It is equal to the original mesh read from the mesh input file. When the **genesis-extent-only** option is activated, an entire genesis-element family hierarchy (i.e., original element plus all descendants) remains together on a processor at all times. In other words, parents and children are never allowed to split among the processors. This does *not* mean that they must remain on the same processor for all time, however. It does mean that if the dynamic load-rebalancing step moves an element, *the entire genesis-element family hierarchy must move with it.*

An example of rebalancing with the genesis-extent-only option is shown in Fig. 2.5. In this example a genesis mesh of four quadrilateral elements is partitioned between two processors. It is assumed that the application code iteratively marks the element in the lower-left corner for refinement. After three refinement iterations, the mesh in the right picture is produced. With the genesis-extent-only option, the children (and grandchildren, etc.) of the original lower-left element cannot be split between the two processors. Therefore, the best the load balancer can do is produce the mesh with 3 elements on processor 0 (P0) and 10 elements on processor 1 (P1). (This is only marginally better than if dynamic load balancing was not performed at all.)

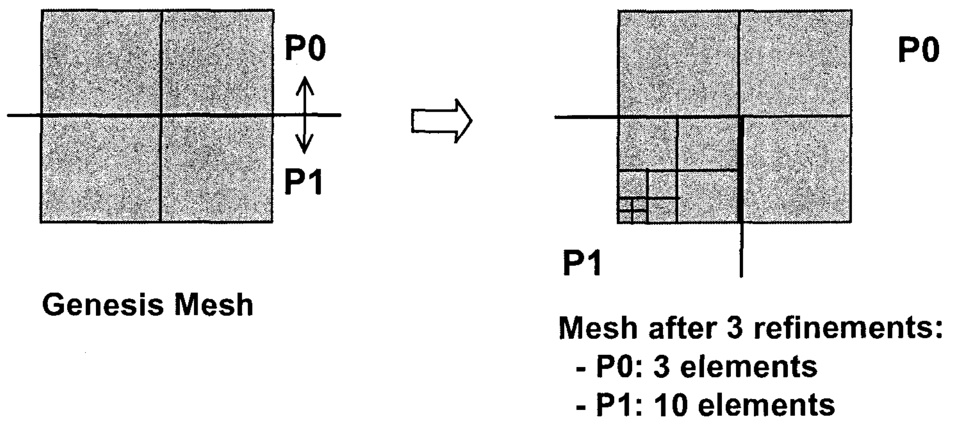

Figure 2.5. Dynamic load rebalancing with the genesis-extent-only option activated.

The trade-off for choosing the genesis-extent-only option is that less communication is required at the expense of a possibly lower-quality partition. The additional communication required for the general load-balancing case (i.e., genesis-extent-only *not* activated) involves parents and off-processor children. If an element is to be unrefined, then its children must all exist on the same processor for that unrefinement to occur. In SIERRA, regardless of the genesis-extent-only option, ***parents with no grandchildren are always kept on the same processor with all of their children***. In other words, the children of these parents are not allowed to move independently of their parent, since these parents are candidates for unrefinement. If grandchildren (and beyond) exist, however, then the children are allowed to independently migrate to another processor (in the general case).

The additional expense for the general case is two-fold: (1) communication lists must be generated and stored for connecting parents with their off-processor children, and (2) any just-unrefined children must be moved back on the same processor as their parent *if* that parent will no longer have any grandchildren following the subsequent refinement step. (This step is to re-unite those children with their parents, a requirement mentioned in the previous paragraph.) The second step involves a lot of bookkeeping and communication since the children of a future one-generation parent (parents without grandchildren following the subsequent refinement) might be spread among several different processors.

For many problems the genesis-extent-only option will likely work well. ***In general, it is recommended that this option be used, and turned off only if it is evident that a sufficiently balanced load cannot otherwise be achieved.*** The example shown in Fig. 2.5 represents a worstcase scenario—refinement into a corner. For the general load-balancing case (genesis-extent-only turned off), a much more balanced mesh is possible. This case is shown in Fig. 2.6. In the picture on the right, three great-grandchildren of the lower-left genesis element have been migrated off P1 onto P0.

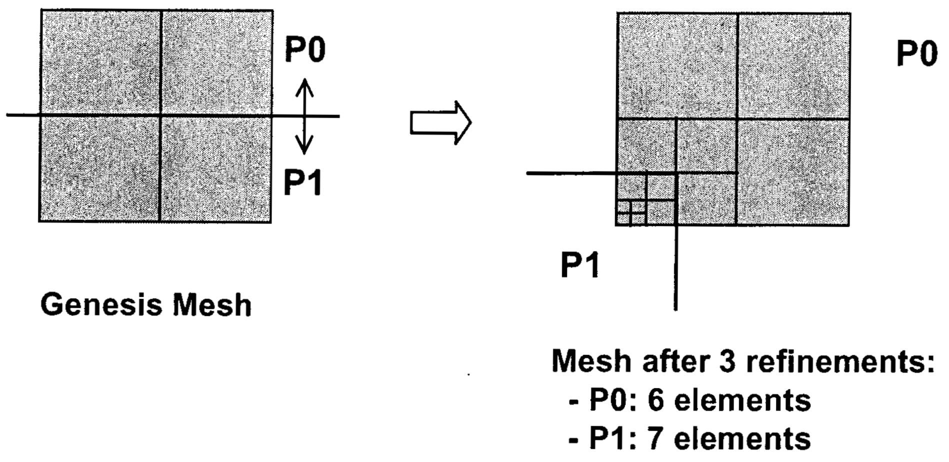

Figure 2.6. Dynamic load rebalancing without the genesis-extent-only option. In this example, a much more balanced mesh is obtained.

3 SIERRA Default *h*-Adaptive Strategy

The *h*-adaptive strategy is the process of converting element error indicators to an element refinement marker (see Fig. 1.1). SIERRA allows for three possible values of this marker: refine, unrefine, or neither. Many adaptive strategies (both *h* and *hp*) have appeared in the literature (see, for example, Refs. 4 and 5), and some are tied to particular error indicators. The default SIERRA strategy is independent of any particular mechanics, and requires user specification of two parameters, α_r and α_u (with $0 \le \alpha_u < \alpha_r \le 1$), which control the amount of refinement and unrefinement, respectively. Given the *global* maximum element error indicator, $e_{\max}$, we compute the refinement and unrefinement error thresholds, viz.,

$$e_{\text{ref}} = \alpha_r e_{\max} \tag{1}$$

$$e_{\text{unref}} = \alpha_u e_{\max} \tag{2}$$

Elements k with an error indicator $e_k > e_{\text{ref}}$ are marked for refinement, while those with $e_k < e_{\text{unref}}$ are marked for unrefinement. All other elements are unmarked (although they still could be refined if necessary for enforcing the maximum-allowable 2:1 refinement-level ratio). It is evident that decreasing α_r increases the amount of refinement, while decreasing α_u decreases the amount of unrefinement. Recall from Section 1 that the element error indicators, e_k, are computed by the application code. A more detailed discussion of the error indicator computation is given in Section 4.1.

4 Responsibilities of Application Codes

To use the adaptivity capability in SIERRA, application codes must supply the following:

- an *a posteriori* error indicator (see Fig. 1.1)
- prolongation and restriction operators for field variables, as required (see Fig. 1.1)
- hanging-node constraints
- execution control code

In addition, the application developer may supply an *h*-adaptive strategy that would override the SIERRA default strategy described in the previous section. The above components are described in more detail in Sections 4.1 through 4.4.

4.1 *A Posteriori* Error Estimation

The role of the *a posteriori* error indicator is to populate an element variable with some physically meaningful "error," which would then be used by the adaptive strategy to mark the element for refinement (if the error is sufficiently large), unrefinement (if the error is sufficiently small), or neither. The error indicator need not be an error *estimator* (which computes an absolute error); it only needs to supply information on the *relative* error distribution among the elements.

Most *a posteriori* error-estimation techniques are strongly dependent on the mechanics and the partial differential equations. Effective error *indicators* also usually require knowledge of the mechanics, such as the use of a first or second derivative of density or Mach number to adapt a mesh in the vicinity of a shock. In many cases the error indicator is a *subset* of the error estimator (e.g., an error indicator might be an element residual or the element contribution to the error measured in some global norm). Therefore, it is impossible for a computational "framework" to provide general error estimators or error indicators to the applications using it. However, SIERRA does provide services that make it easier to develop these estimators or indicators, especially in a parallel environment. Hereafter, we drop the term "error indicator" and use only "error estimator" for simplicity.

In many cases it is necessary to form a local element patch for computing the error estimate. Such a patch might be formed of elements containing a particular node (a node-based patch) or of elements connected to a particular element (an element-based patch). Examples of patch-based error estimators are Zienkiewicz-Zhu (ZZ) [6] and subdomain residual methods [7]. The dynamic creation of these patches is handled by SIERRA for a given node or element object (see `Fmwk_MeshObj.h`). In order to use this functionality, the application must first instruct SIERRA to assemble and store the node-to-element connectivity. If the application wants to preserve memory and not store those connectivities, the patches can alternatively be assembled (directly by the application) by iterating the standard element-to-node connectivities. In a parallel environment the situation is more complicated, as shown in Fig. 4.1. The formation of patches for nodes or elements on a processor boundary requires communication to neighboring processors. In this case, (temporary, read-only) ghost objects (nodes and elements) must be constructed before the patches can be created. All communication and construction of the ghost objects are handled by SIERRA. The application is responsible for instructing SIERRA to create the ghost objects (see `Fmwk_MeshManufacture.h`), and also for deleting the ghost objects when they are no longer needed. (The capability for *persistent* (readable *and* writeable) ghosting of mesh objects is planned for future versions of SIERRA. In this case the framework would handle the deleting of ghost objects, as well as the maintaining of globally consistent updates of their field values, the reghosting of objects following dynamic load rebalancing or restart, etc.)

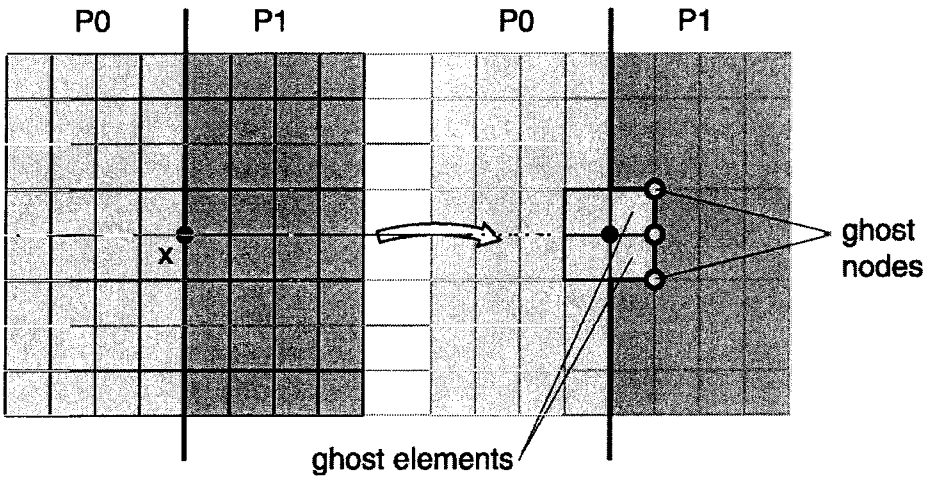

Figure 4.1. Generation by SIERRA of a node-based element patch for node x. The node lies on an interprocessor boundary (see left picture). To create this patch on processor 0, two elements and three nodes are "ghosted" (see right picture). These ghost objects are temporary read-only *copies* that can be discarded (by processor 0) when the patch is no longer needed.

SIERRA provides many utility functions for helping the application compute an error estimate. These include functions to compute local (on a given processor) and global (across all processors) L_2 norms, as well as local and global element-error minima and maxima. Specifically for the ZZ error estimator, SIERRA provides a function to register (i.e., tell the framework to allocate memory for) the recovered nodal gradient on all the mesh nodes. Also provided is a function to recover the gradient of a scalar nodal variable (see `Apub_RecoverGradient.h`); the function includes the local least-squares projection (currently only the projection onto a trilinear polynomial has been implemented). The application needs to provide only the values of the scalar gradient and the $(1, x, y, z, xy, yz, xz, xyz)$ trilinear basis functions at the sampling points in each element.

Many finite-element error estimators require the solution of local residual problems, a global dual problem, or both (see, for example, Refs. 5, 7, 8, and 9). While these problems involve the solution of partial differential equations and therefore must be solved by the mechanics application, SIERRA does provide services for setting up and handling the data for those problems. The *a posteriori* error bounds developed by Paraschivoiu et al. [8], for example, require the solution of three new problems (in addition to the primary problem): (1) a series of local *primal* residual problems on a *broken* "truth" mesh, (2) a global dual problem on the original coarse mesh, and (3) a series of local *dual* residual problems on a broken "truth" mesh. Fig. 4.2 shows how the broken "truth" mesh is obtained from the original coarse mesh.

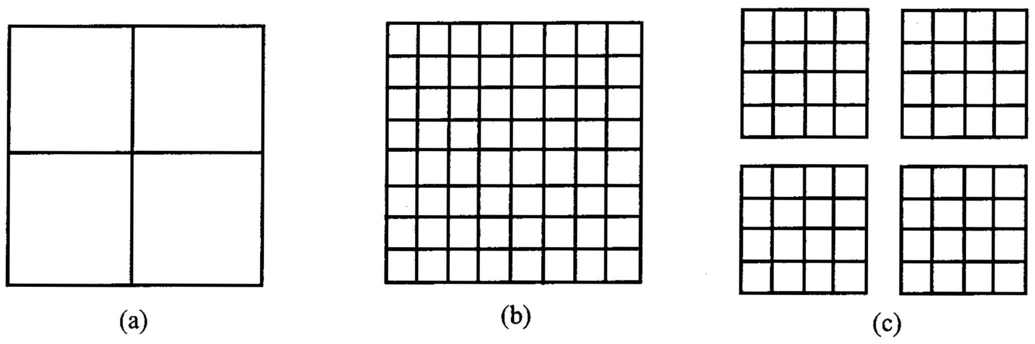

Figure 4.2. Obtaining the broken "truth" mesh. Definitions of (a) an original coarse "working" mesh; (b) a "truth" mesh, which is a global refinement of the coarse working mesh; and (c) the *broken* "truth" mesh, where each piece is a refinement of a single element in the coarse working mesh. The *a posteriori* error estimator described in Ref. 8 solves local residual problems independently on each piece of the broken "truth" mesh.

One way this may be addressed in an application is by creating a new SIERRA region (see Ref. 10 for a description of the SIERRA region) for each different problem. This leads to the coexistence of four regions: the coarse *primal* region (i.e., the original region used by the application), the "truth" *primal* region, the coarse *dual* region, and the "truth" *dual* region. Each region has its own mesh and fields, but the mechanics algorithms can be shared among regions if necessary. For example, if the differential operator is self-adjoint, the left-hand-side computation for the primal and dual problems will be the same and, therefore, use the same mechanics algorithms. Eventually, SIERRA will provide a capability to create these new regions through a pseudocopy operator, allowing the application to include or exclude algorithms, insert a different mesh, etc., into the "copied" region. (This operator is not available in SIERRA Framework Version 3.) Also, the SIERRA transfer algorithms [10,11] can be used to exchange data between the regions. This is needed, for example, for computing the right-hand-side residuals for the primal and dual "truth" problems.

Another class of error estimators is based on *extrapolation*, such as Richardson extrapolation. Implementation of these types of error estimators is made much easier with the SIERRA adaptivity services, particularly the mesh refinement capabilities coupled with the application's prolongation operators. The SIERRA transfer operators would also be useful if extrapolation to arbitrarily sized meshes is desired (i.e., meshes that are not obtained by subdividing coarse elements).

4.2 Prolongation and Restriction Operators

The prolongation and restriction operators are used to assign new values to field variables on the newly refined and unrefined mesh objects, respectively. For nodal variables, prolongation often involves simply averaging the field values of its parent object's (edge, face or element) nodes. There is public code for managing this (in the `apublic` area of SIERRA) that can be called by any application. Care must be taken whenever the new node is placed on a nonplanar surface or material interface and snapped to the actual geometry. (As mentioned in Section 2.1, this capability is not available as of the release of SIERRA Framework Version 3.01, but is planned for the future.) Since the node's location, in general, would not be at the centroid of its parent face or edge, special treatment might be necessary. In most cases, no action is required for nodal variables following unrefinement because the parent nodes are the same ones used in the child elements. In those instances the nodal restriction operator is a no-op.

Some variables may require a particular type of interpolation (e.g., mass-conserving) following refinement, while other variables, such as an element error norm, would not require any action (since the error norm on the child elements would be recomputed following the subsequent computation on the adapted mesh). The possibilities for prolongation and restriction algorithms, particularly for element variables, are many. They depend strongly on the specific variable as well as on the application. Some very simple algorithms are described below.

Copy prolongation operator

This operator might typically be applied for element variables on a uniform strain element. In this operator, shown in Fig. 4.3, all element values in the parent element are copied to the child elements.

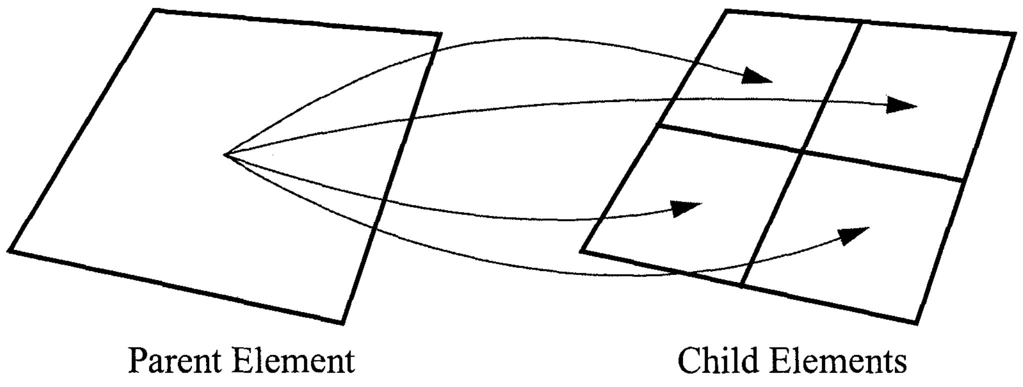

Figure 4.3. Copy prolongation operator.

Copy nearest integration-point prolongation operator

This operator might typically be applied on fully integrated elements for element variables defined at integration points. As depicted in Fig. 4.4, integration-point values in the parent element are copied to all of the integration points in the child element that contains the parent integration point. (We assume here that each child element contains exactly one of its parent's integration points.)

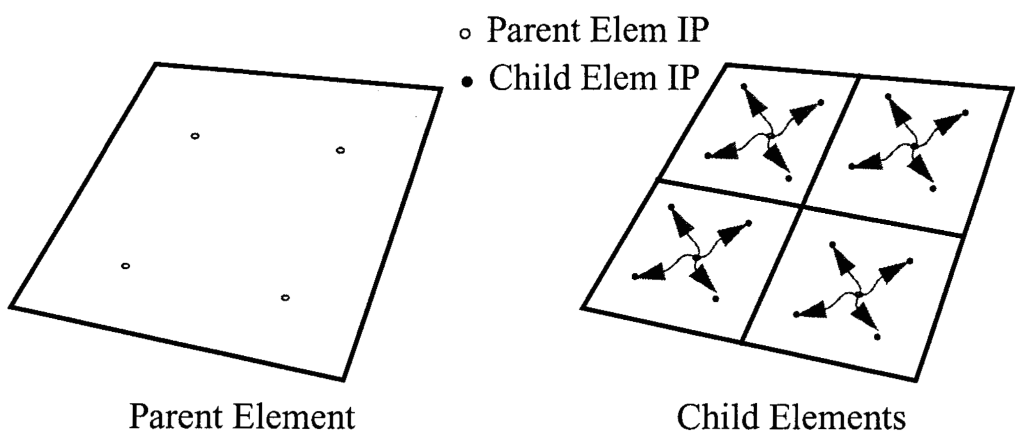

Figure 4.4. Copy nearest integration-point prolongation operator.

Interpolate prolongation operator

This operator might typically be applied on fully integrated elements for element variables defined at integration points. As depicted in Fig. 4.5, integration-point values in the parent element are used to calculate a least-squares fit to a linear interpolation. The child integration-point values are then interpolated. This operator can also be applied to uniform strain elements. In this case, the least-squares interpolation function is computed on an element-centered patch with respect to the parent element. The quasi-statics code *adagio* [12] uses this type of prolongation operator for the following element variables: stress, rotated stress, and strain energy. The method used to generate the least-squares function is exactly the same as that used for the ZZ error estimator (see Section 4.1). Note: In a Lagrangian context, this must be recomputed every increment if the geometry changes.

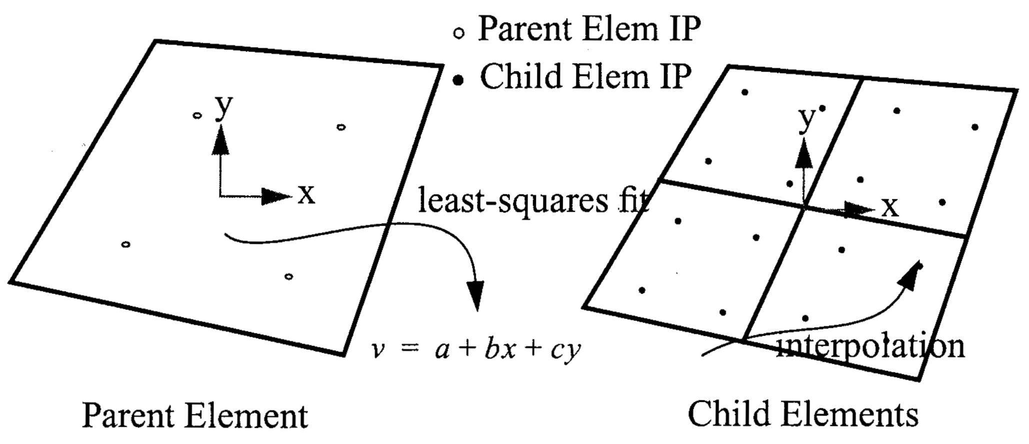

Figure 4.5. Interpolate prolongation operator.

Average restriction operator

This operator is typically applied to a uniform strain formulation element. All element values in the child elements are averaged and the results copied to the parent element, as shown in Fig. 4.6.

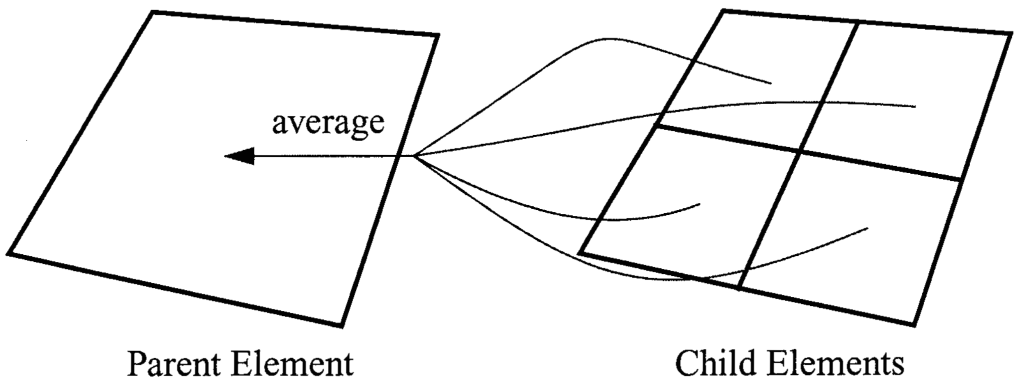

Figure 4.6. Average restriction operator.

Average nearest integration-point restriction operator

This operator is typically applied on fully integrated elements for element variables defined at integration points. Integration-point values in a child element are averaged and copied to the integration point in the parent element that is contained inside that child element. This is depicted in Fig. 4.7.

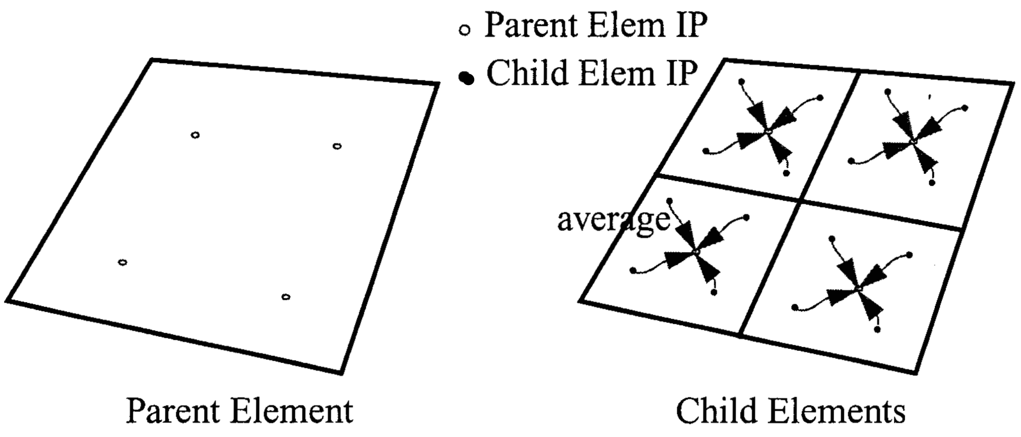

Figure 4.7. Average nearest integration-point restriction operator.

Interpolate restriction operator

This operator is typically applied on fully integrated elements for element variables defined at integration points. Integration-point values in the child element are used to calculate a least-squares fit to a linear interpolation. The values associated with the parent integration point that falls in that child element are then interpolated. This is depicted in Fig. 4.8.

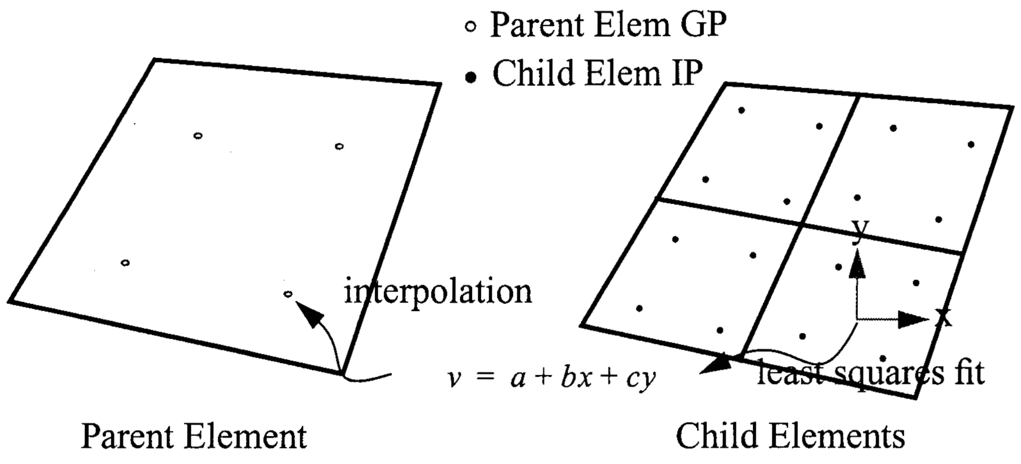

Figure 4.8. Interpolate restriction operator.

4.3 Hanging-Node Constraint Handling

Hanging nodes are created along the transition regions between refinement levels in a mesh. A simple example is shown in Fig. 4.9. One of two hexahedra is refined, which creates the five hanging nodes shown in the picture on the right. One hanging node is on the face that separates the two elements, and the other four are at the midpoints of edges. (It is important to note that SIERRA does not actually generate these internal faces and edges unless the application code requests them.) The application developers are responsible for writing the equations to handle this situation.

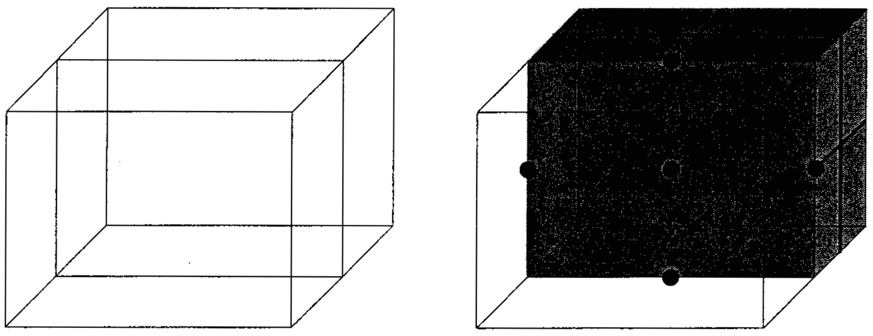

Figure 4.9. Creation of hanging nodes following adaptive refinement.

The simplest approach is to constrain the value of any field defined on the node to be the average of the nodal values of its parent object. For example, the face hanging node in Fig. 4.9 has that face as its parent. The constraint would force that node's fields to be the average of the values of the face's four corner nodes (this makes sense because the hanging node is at the center of the face). Similarly, the edge hanging-node field values would be constrained to be the average of the values of the edge's two nodes.

The framework provides services to assist application developers in writing the constraint equations. These services include providing a list of all hanging nodes, simple access to the nodes of the parent object (edge or face), and a way to iterate those nodes.

Application developers are also responsible for enforcing the hanging-node constraints. If linear solvers are used, enforcement may be possible through a particular linear-solver package. Constraints may be enforced, for example, with Lagrange multipliers or a penalty formulation. Finally, application developers are responsible for resolving possible conflicts between constraints. This might happen if a hanging node is on a Dirichlet boundary, or if a hanging node is involved in contact.

4.4 Execution Control

The execution control code includes the calls to the SIERRA methods that perform the mesh refinement and unrefinement. An example containing pseudocode is shown in Fig. 4.10. The code corresponds very closely with the algorithm flow diagram given in Fig. 1.1.

The control code also contains the logic, such as looping constructs, that might dictate, for example, how many times the sequence of mesh-adaptation steps gets executed for a single time step in a transient calculation. Following a single mesh-adaptation sequence, the user may wish to re-solve for the physics solution on the new mesh (the *outer adapt loop*), or to speed up execution (at the expense of accuracy of the error indicator), the user may wish to repeat the adaptation sequence using the prolongated/restricted solution, without re-solving (the *inner adapt loop*). These kinds of decisions are application driven.

The "MARK ELEMENTS" step in Fig. 4.10 refers to the *h*-adaptive strategy. The default strategy (supplied by the framework) was described in Section 3. The framework version is a C++ virtual method (in the `Fmwk_HAdapt` class) that can be overloaded by a derived application class.

```
time loop {
    outer adapt loop {
        solve physics
        inner adapt loop {
            compute error indicator
            if (stopping criterion) break
            MARK ELEMENTS                 // Framework or application
            RESOLVE MARKERS               // Framework
            restrict variables
            GLOBAL UPDATE MESH            // Framework
            prolong variables
            MESH UPDATE COMPLETION        // Framework
        }
    }
}
```

Figure 4.10. Example of execution control (pseudo) code driving the adaptivity process. Code such as this must be supplied inside the application.

The stopping criterion shown both in Fig. 1.1 and Fig. 4.10 is also supplied by the application code. Usually a stopping criterion would be error-based—if the error (as computed by the error estimator) is below some tolerance, execution would break out of the adapt loop and proceed to the next time step. If such a stopping criterion is not implemented, however, the alternative is to specify (possibly via the input file) a specific number of inner and outer iterations to be executed at each time step. In the absence of a reliable error estimator, this is perhaps the only alternative.

References

1. H. C. Edwards. *SIERRA Framework Version 3: Core Services Theory and Design.* SAND2002-3616. Albuquerque, NM: Sandia National Laboratories, 2002.

2. M. W. Beall and M. S. Shephard. "An Object-Oriented Framework for Reliable Numerical Simulations." *Engineering with Computers* 15, no. 1 (1999): 61–72.

3. K. Devine, B. Hendrickson, E. Boman, M. St. John, and C. Vaughan. *Zoltan: A Dynamic Load-Balancing Library for Parallel Applications - User's Guide.* SAND99-1377. Albuquerque, NM: Sandia National Laboratories, 1999.

4. P. Diez and A. Huerta. "A Unified Approach to Remeshing Strategies for Finite Element h-Adaptivity." *Computer Methods in Applied Mechanics and Engineering* 176 (1999): 215–229.

5. J. R. Stewart and T. J. R. Hughes. "An A Posteriori Error Estimator and hp-Adaptive Strategy for Finite Element Discretizations of the Helmholtz Equation in Exterior Domains." *Finite Elements in Analysis and Design* 25 (1997): 1–26.

6. O. C. Zienkiewicz and J. Z. Zhu. "A Simple Error Estimator in the Finite Element Method." *International Journal for Numerical Methods in Engineering* 24 (1987): 337–357.

7. I. Babuska and T. Strouboulis. *The Finite Element Method and its Reliability.* Oxford, UK: Oxford University Press, 2001.

8. M. Paraschivoiu, J. Peraire, and A. T. Patera. "A Posteriori Finite Element Bounds for Linear-Functional Outputs of Elliptic Partial Differential Equations." *Computer Methods in Applied Mechanics and Engineering* 150 (1997): 289–312.

9. D. Estep, M. G. Larson, and R. D. Williams. "Estimating the Error of Numerical Solutions of Systems of Reaction-Diffusion Equations." *Memoirs of the American Mathematical Society* 146, no. 696 (July 2000).

10. J. R. Stewart and H. C. Edwards. "The SIERRA Framework for Developing Advanced Parallel Mechanics Applications." In *Proceedings of the First Sandia Workshop on Large-Scale PDE-Constrained Optimization*, Santa Fe, NM, April 4–6, 2001, edited by O. Ghattas, Springer's Lecture Notes in Computational Science and Engineering, 2001.

11. J. R. Stewart, W. R. Witkowski, K. D. Copps, H. C. Edwards, and J. D. Zepper. "Advanced Technologies for Parallel Adaptive Multiphysics Simulation." In *Proceedings of the Fifth World Congress on Computational Mechanics (WCCM V)*, Vienna, Austria, July 7–12, 2002, edited by H. A. Mang, F. G. Rammerstorfer, and J. Eberhardsteiner. Vienna, Austria: Vienna University of Technology. ISBN 3-9501554-0-6. Available at http:// wccm.tuwien.ac.at.

12. J. A. Mitchell, A. S. Gullerud, W. M. Scherzinger, R. Koteras, and V. L. Porter. "Adagio: Non-Linear Quasi-Static Structural Response Using the SIERRA Framework." In

Proceedings of the First MIT Conference in Computational Fluid and Structural Mechanics, 361–364. Amsterdam: Elsevier, 2001.

Distribution

External

Texas Institutute for Computational and Applied Mathematics
University of Texas at Austin
Austin, TX 78712
Attn: J. Tinsley Oden

Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, CA 94551-0808
Attn: Evi Dube

Los Alamos National Laboratory
P.O. Box 1663, MS F652
Los Alamos, NM 87545
Attn: James S. Peery

Massachusetts Institute of Technology
77 Massachusetts Avenue, Room 37-451
Cambridge, MA 02139
Attn: Jaime Peraire

Colorado State University
Department of Mathematics
101 Weber Building
Fort Collins, CO 80523-1874
Attn: Donald Estep

Rensselaer Polytechnic Institute
110 8th St.
Troy, NY 12180
Attn: Joseph E. Flaherty

Internal

1	MS 0841	9100	T. C. Bickel
1	MS 0835	9140	J. M. McGlaun
5	MS 0835	9113	S. N. Kempka
10	MS 0827	9143	J. D. Zepper
1	MS 0824	9110	A. C. Ratzel

1	MS 0421	9800	W. L. Hermina,
1	MS 0834	9114	J. E. Johannes
1	MS 0836	9115	E. S. Hertel
1	MS 0847	9120	H. S. Morgan
1	MS 0824	9130	J. L. Moya
1	MS 0828	9133	M. Pilch
1	MS 0847	9211	S. A. Mitchell
1	MS 1110	9214	D. E. Womble
1	MS 0819	9231	E. A. Boucheron
1	MS 0139	9900	M. O. Vahle
1	MS 0835	9141	S. W. Bova
1	MS 0835	9141	R. J. Cochran
1	MS 0835	9141	S. P. Domino
1	MS 0835	9141	M. W. Glass
1	MS 0835	9141	R. R. Lober
1	MS 0835	9141	A. A. Lorber
1	MS 0835	9141	P. A. Sackinger
1	MS 0835	9141	J. H. Strickland
1	MS 0835	9141	S. R. Subia
1	MS 9217	8920	C. J. Aro
1	MS 9042	8728	C. D. Moen
1	MS 0826	9113	D. R. Noble
1	MS 0826	9114	E. S. Piekos
1	MS 0834	9114	M. M. Hopkins
1	MS 0834	9114	P. K. Notz
1	MS 0825	9115	J. L. Payne
1	MS 0838	9116	R. E. Hogan
1	MS 0828	9133	K. J. Dowding
1	MS 0847	9133	W. R. Witkowski
1	MS 0316	9233	C. C. Ober
1	MS 0316	9233	T. M. Smith
1	MS 0316	9233	R. Hooper
1	MS 0847	9142	M. K. Bhardwaj
1	MS 0847	9142	M. L. Blanford
1	MS 0847	9142	A. S. Gullerud
1	MS 0835	9142	J. D. Hales
1	MS 0847	9142	M. W. Heinstein
1	MS 0847	9142	S. W. Key
1	MS 0847	9142	W. S. Klug
1	MS 0847	9142	J. R. Koteras
1	MS 0847	9142	N. K. Crane

1	MS 0847	9142	J. A. Mitchell
1	MS 0835	9142	K. H. Pierson
1	MS 0847	9142	V. L. Porter
1	MS 0847	9142	T. J. Preston
1	MS 0847	9142	G. M. Reese
1	MS 0847	9142	T. F. Walsh
1	MS 0807	9338	B. H. Cole
1	MS 0847	9142	K. F. Alvin
1	MS 9217	9214	M. F. Adams
1	MS 0847	9127	J. Jung
1	MS 9405	8726	R. E. Jones
1	MS 0847	9211	M. S. Eldred
1	MS 0827	9143	K. M. Aragon
1	MS 0827	9143	K. N. Belcourt
1	MS 0827	9143	D. M. Brethauer
1	MS 0827	9143	K. D. Copps
20	MS 0827	9143	H. C. Edwards
1	MS 0827	9143	C. A. Forsythe
1	MS 0827	9143	M. E. Hamilton
1	MS 0827	9143	J. R. Overfelt
1	MS 0827	9143	J. S. Rath
1	MS 0827	9143	G. D. Sjaardema
20	MS 0827	9143	J. R. Stewart
1	MS 0827	8920	A. B. Williams
1	MS 1111	9215	K. D. Devine
1	MS 0819	9231	K. H. Brown
1	MS 0819	9231	K. G. Budge
1	MS 0819	9231	S. P. Burns
1	MS 0819	9231	D. E. Carrol
1	MS 0819	9231	R. R. Drake
1	MS 0847	9226	S. J. Owen
1	MS 9018	8945-1	Central Technical Files
2	MS 0899	9616	Technical Library
1	MS 0612	9612	Review & Approval Desk for DOE/OSTI